Interactive Synthesis using Free-Form Queries

Tihomir Gvero and Viktor Kuncak
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
Email: firstname.lastname@epfl.ch

***Abstract*—We present a new code assistance tool for integrated development environments. Our system accepts free-form queries allowing a mixture of English and Java as an input, and produces Java code fragments that take the query into account and respect syntax, types, and scoping rules of Java as well as statistical usage patterns. The returned results need not have the structure of any previously seen code fragment. As part of our system we have constructed a probabilistic context free grammar for Java constructs and library invocations, as well as an algorithm that uses a customized natural language processing tool chain to extract information from free-form text queries. The evaluation results show that our technique can tolerate much of the flexibility present in natural language, and can also be used to repair incorrect Java expressions that contain useful information about the developer's intent. Our demo video is available at http://youtu.be/tx4-XgAZkKU**

I. Introduction

Application programming interfaces (APIs) are becoming more and more complex, presenting a bottleneck when solving simple tasks, especially for new developers. APIs contain many types and declarations, so it is difficult to know how to combine them to achieve a task of interest. Instead of focusing on more creative aspects of the development, a developer ends up spending a lot of time trying to understand informal documentation or adapt the API documentation examples. Integrated development environments (IDEs) help in this task by listing declarations that belong to a given type, but leave it to the developer to decide how to combine the declarations.

On the other hand, on-line repository host services such as GitHub [1], BitBucket [2] and SourceForge [3] are becoming more and more popular, hosting a large number of freely accessible projects. Such repositories are an excellent sources of code examples that developers can use to learn API usage. Moreover, their large size and variety suggests that they can be used by machine learning techniques to create more sophisticated IDE support. A natural first step is to perform code search [4], though this still leaves the user with the task of understanding the context and adapting it to their needs. Several researchers have pursued the problem of generalizing from such examples in repositories, combining non-trivial program analysis and machine learning techniques [5].

In this paper, we present a new approach that synthesizes code appropriate for a given program point, guided by hints given in *free-form* text. We have implemented our approach in a system *anyCode*, and have found it to be very useful in our experience (see our technical report [6, Section 4]). Our approach builds a model of the Java language based on corpus of code in repositories, and adapts the model to a given text input. In that sense, our approach combines some of the advantages of statistical programming language models [5] but also of natural language processing of the input containing English phrases, previously done for restricted APIs [7].

Our approach builds on our past experience with the InSynth tool [8], in which a user only indicates the desired API type; the tool InSynth then generates ranked expressions of a given type. With our new tool, *anyCode*, the input can be interpreted as a result type, but, more generally, it can be any text, referring to any part of an expected expression. We find this interface more convenient and expressive than in InSynth. Furthermore, InSynth uses only the unigram model [9, Chapter 4], which assigns a probability to a declaration based on its call frequency in a corpus. InSynth uses the model to synthesize and rank expressions. In *anyCode*, besides unigram, we use the more sophisticated probabilistic context free grammar (PCFG) model [9, Chapter 14], to synthesize and rank the expressions. We perform synthesis in three phases: (1) we use natural language processing (NLP) tools [10]–[12] to structure the input text and split the text input into chunks of words based on their relationships in the sentence parse tree; (2) we use the structured text and unigram model to select a set of most likely API declarations, using a set of scoring metrics and the Hungarian method [13] to solve the resulting assignment problem [14]; (3) we finally use the selected declarations, PCFG and unigram model to unfold the declaration arguments. The result is a list of ranked (partial) expressions, which *anyCode* offers to the developer using the familiar code completion interface of Eclipse.

By introducing a textual input interface, we aim to automatically reduce the gap between a natural and a programming language. *anyCode* allows the developer to formulate a query using a mixture of English and code fragments. *anyCode* takes into account English grammar when processing input text. To improve the input flexibility and expressiveness we also consider word synonyms and other related words (hypernyms and hyponyms), and build a related word map based on WordNet [15], a large lexical database of English. The techniques we implement in *anyCode* are inspired by stochastic machine translation. However, we had to overcome the lack of a parallel corpus relating English and Java, as well as the gap between an informal medium such as English and the rigorous syntax and type rules of a programming language such as Java.

We aim to relieve the user of the strict structure of a programming language when describing their intention. From our perspective, IDE tools should allow a user to gloss over aspects such as the number and the order of arguments in

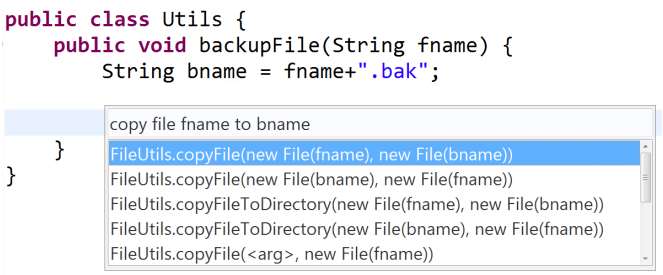

Fig. 1. After the user inserts text input, *anyCode* suggests five highest-ranked well-typed expressions that it synthesized for this input.

method calls, or parenthesis usage. Instead, the developers should focus more on solving important higher-level software architecture and decomposition problems. Finally, we also hope to lower the entry for those who are learning to program, for whom syntax is often one of the first obstacles. To achieve this, we find that a short text input that approximately describes the structure of the desired expression is the most convenient. To make the input useful for programming, we also allow a user to explicitly write literals and local variable names in input. Using such input, *anyCode* manages to synthesize valid Java code fragments. It can do that because it does not impose any strict requirement on the input: it has the ability to generate likely expressions according to the Java language model, and uses as much of the information from the input as it can extract to steer the generation towards developer's intention.

II. Examples

In this section we use four examples to demonstrate main functionalities of *anyCode*.

1) Making a File Backup. Suppose that a user wants to implement a method that backs up the content of a file. The method should take a file name as a parameter and copy the content of the file to a new file with an appropriately modified name. First, a user writes an incomplete code that takes the parameter fname with the file name, as shown in Figure 1. She also creates a variable bname that stores the backup file name, that is obtained by adding ".bak" extension to bname. In the next line, instead of writing the code, the user can invoke *anyCode*. When *anyCode* is invoked, a pop-up text field appears where a user can insert the text, such as 'copy file fname to bname' that specifies her desire to copy the file content. *anyCode* automatically extracts the program context from Eclipse and identifies words fname and bname in the input as values referring to a parameter and a local variable. *anyCode* then uses this information to generate and present several ranked expressions to the user. When the user makes her choice, the tool inserts the chosen expression at the invocation point. In this example, *anyCode* works for less than 50 milliseconds and then presents five solutions of which the first one copies the file fname content to a file with name bname: FileUtils.copyFile(**new** File(fname), **new** File(bname)). This is a valid solution; it uses the method FileUtils.copyFile from the popular "Commons IO" library.

2) Invoking the Class Loader. Suppose that a user intends to load a class with a name "MyClass.class". She invokes the tool with the free-form input 'load class "MyClass.class"' In less than 40 milliseconds, *anyCode* automatically synthesizes and suggests the following (partial) expressions:

```
1  Thread.currentThread().getContextClassLoader()
                         .loadClass("MyClass.class").getClass()
2  Thread.currentThread().getContextClassLoader()
                         .loadClass("MyClass.class")
3  Thread.currentThread().getContextClassLoader().loadClass(⟨arg⟩)
                         .getClass()
4  "MyClass.class".getClass()
5  Thread.currentThread().getContextClassLoader().loadClass(⟨arg⟩)
```

The second suggestion turns out to be the desired one. Note that the suggestions 1, 2, and 4 represent complete expressions. On the other hand, suggestions 3 and 5 represent templates that include the symbol ⟨arg⟩ that marks the places where local variables are often used. The main reason why we present templates is that a user often inserts incomplete input and for an incomplete input the best solution is an incomplete output, i.e., a template. If we have insisted only on completed expressions, we would miss many interesting solutions that are more convenient for incomplete inputs.

Note that the complete expressions 1 and 2 include declarations whose selection and integration does not directly depend on the textual input. For instance, method loadClass contains both input words load and class, whereas currentThread does not. To reach the currentThread from loadClass we use probabilistic language model for Java and its API calls, derived from a corpus of code. Without such a model we would not be able to construct complex expressions such as the above one.

3) Creating a Temporary File. In the third example we demonstrate the use of semantically related words. For instance, if a user wants to discover templates that make a new file, she may insert 'make file'. In a less than 80 milliseconds, *anyCode* generates the following output:

```
1  new File(⟨arg⟩).createNewFile()
2  new File(⟨arg⟩).isFile()
3  new File(⟨arg⟩)
4  new FileInputStream(⟨arg⟩)
5  new FileOutputStream(⟨arg⟩)
```

Note that word make does not appear among the solutions, because API designers used the word create. *anyCode* succeeds in finding the solution because it considers, in addition to the words such as make appearing in the input, its related words, which includes create. *anyCode* uses a custom related-word map to compute the relevant words. We built this map by automatically processing and adapting WordNet, a large lexical semantic network of English words.

4) Reading from a File. In the final example we show that our input interface may also accept an approximate expression. For instance, if a user attempts to write an expression that reads the file, in the first iteration she may write the expression 'readFile("text.txt","UTF−8")'. Unfortunately, this expression is not well-typed according to common Java APIs. Nevertheless, if *anyCode* takes such a broken expression, it pulls it apart and

recomposes it into a correct one, suggesting (again in less than 40 milliseconds) the following solutions:

```
1  FileUtils.readFileToString(new File("text.txt"))
2  FileUtils.readFileToString(new File("UTF−8"))
3  FileUtils.readFileToString(⟨arg⟩)
4  FileUtils.readFileToString(new File(⟨arg⟩))
5  FileUtils.readFileToString(new File("text.txt"), "UTF−8")
```

anyCode first transforms the input by ignoring the language specific symbols (e.g., parenthesis and commas). It then slices complex identifiers, so called *k-words*, into single words. Here, readFile is a 2-word that gets sliced into read and file. Despite the loss of some structure in treating the input, our language model gives us the power to recover meaningful expressions from such an input. This shows that *anyCode* can be used as a simple expression repair system. The desired solution is ranked fifth because it uses a version of readFileToString method with two arguments, which appears less frequently in the corpus than the simpler versions of the method.

III. System Overview

In this section we give a high-level picture of the main components of *anyCode*. Input to our *anyCode* consists of i) a textual description, explicitly typed by the developer and ii) a partial Java program with a position of the cursor, which *anyCode* extracts automatically from the Eclipse IDE. *anyCode* uses the input to generate, rank and present (possibly partial) expressions to the user. As Figure 2 shows, the key components of *anyCode* are the text parser, the declaration search engine, and the expression synthesizer. The method getExpressions performs these steps, as outlined in Figure 3.

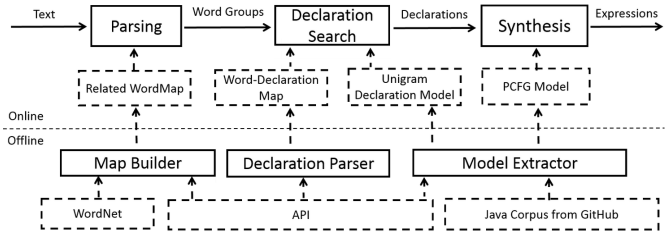

Fig. 2. *anyCode* system overview. The offline components run only once and for all. The online components run as part of the Eclipse plugin.

The parsing identifies structure of the input text using a set of natural language processing tools. *anyCode* uses the structure to group input words into WordGroups. The intuition is that the input text corresponds to several declarations, and grouping according to the rules of English helps to identify these declarations. Moreover, the system uses a map of related words to complete the words given in the input with some of the related meanings computed from a modified version of WordNet [15]. To complement natural language input, the system uses program context from the IDE to mark local variables and literals in the input text.

Next, the declaration search engine uses WordGroups, to find a subset of API declarations that are most likely to

```
getExpressions(text, context, NBestExprs):
  // Text Parsing
  (WordGroups, Literals, Locals) ← parse(text, context)
  // Declaration Search
  DeclGroups ← declSearch(WordGroups, API, Unigram)
  // Synthesis
  ExPCFG ← extend(PCFG, Literals, Locals)
  Exprs ← synth(DeclGroups, ExPCFG , NSteps)
  return keepBest(Exprs, NBestExprs)
```

Fig. 3. The high level description of online portion of *anyCode*.

form the final expressions. The system tries to match word groups against declarations in our API collection. To perform matching, the system extracts a list of words from declarations and matches them against the words in the groups. Based on the number of words that match the system estimates the matching score. Together with the declaration Unigram [9, Chapter 4] score it forms the final declaration score. Finally, for each word group the system selects the top NBest declarations with the highest final scores and puts them in a declaration group. In summary, the method declSearch transforms each word group into a declaration group. Our API collection contains declarations we collect from several APIs. It is organized as a word-declaration map that maps words to declarations and precomputed to speed up the selection.

Finally, the synthesis uses declaration groups and a probabilistic context free grammar (PCFG) model [9, Chapter 14] to generate expression. The production rules of PCFG encode declaration compositions. ExPCFG consists of the initial PCFG model, pre-extracted from a Java source code corpus (including 14'500 projects from GitHub), and the extension, production rules for literals and local variables, extracted from the partial program. The method synth tries to unfold declaration arguments following ExPCFG model, in NSteps. Given a declaration, ExPCFG suggests declarations that should fill in the argument places. The method synth also assigns scores to the expressions, based on the ExPCFG and declarations scores. It returns the best NBestExprs expressions.

IV. Evaluation

We have evaluated our system on 45 examples. In our technical report [6], Figure 2 shows for each example a free-form query and the code that we expected to obtain in return. The results show that *anyCode* can efficiently synthesize the expressions in a small period of time (in less that 200 milliseconds). They also show that *anyCode* without the both models generates only 6 expected expressions among the top ten solutions. *anyCode* with unigram model generates 19 and with both models generates all 45 expressions.

V. Limitations

The first limitation is related to our set of examples. While fairly large by the standards of previous literature, it may not be representative of general results. This limitation comes from the two facts: (1) there is no standardized set of benchmarks

for the problem that we examine, and (2) we used the same set of examples to configure and evaluate our system. The primary purpose of the examples is to show that our tool is able to produce a set of real-world examples when configured. A parallel corpus with text as input and declarations (expressions) as output would be ideal for configuring the parameters, however no such corpus exists. Our attempt to create the corpus from the code and its descriptive comments led to irrelevant examples and the low quality corpus. The second limitation is related to the complexity of the code snippets we synthesize. It comes from the fact that we synthesize only expressions, excluding local variable declarations and other statements. We plan to improve *anyCode* and to overcome these limitations in our future work.

VI. Related Work

There are many tools that suggest code fragments [4], [5], [16]–[20]. Prospector [21] uses a receiver type $type_{in}$ to generate a chain of method calls that ends with a method whose return type is equal to the desired type $type_{out}$. It ranks the solution by the length, preferring the shorter ones. To fill in the method arguments, and produce a complete expressions, a user often needs to initiate multiple queries. Unlike Prospector, *anyCode* uses statistics from the corpus to synthesize and rank expressions and it requires a single invocation to produce complete expressions.

There are also many tools that suggest snippets using natural language input. SmartSynth [7] generates smartphone automation scripts from natural language descriptions. Macho [22] transforms natural language descriptions into a simple programs using a natural language parser, a corpus and input-output examples. Little and Miller [23] built a system that translates keywords into a valid expression. NaturalJava [24] allows a user to create and manipulate Java programs using an NL input. SNIFF [25] uses natural language to search for code examples. The tools [26], [27] search and synthesize code fragments using input-output examples. Finally, the tool [28] uses genetic programming approach with different degrees of human guidance which includes names of library functions and test cases. The tool grows a new functionality using user's suggestions and grafts it to the existing code. Unlike mentioned tools, *anyCode* is the first tool that uses free-form queries, and combines NL tools, PCFG and unigram model to synthesize, repair and rank expressions. *anyCode* can also synthesize previously unseen code fragments. It also *automatically* infers the set of words that map to declaration (components). Moreover, *anyCode* uses a sophisticated model that maps input to declarations, resolves complex declaration names and takes into account related words (e.g., synonyms).

VII. Conclusions

We presented *anyCode*, a synthesis tool that combines unique flexibility in both its input, that may contain a mixture of English and code fragments, and its output, that may include combinations of declarations not encountered previously in the corpus. Our experience with the tool, suggests that there is a number of scenarios where such functionality can be useful for the developer.

Acknowledgment

We thank Rastislav Bodik, Armando Solar-Lezama, Darko Marinov and Ravichandhran Kandhadai Madhavan for the valuable discussions and comments. This work is supported by the ERC project "Implicit Programming".

References

[1] GitHub repository hosting service, https://github.com/.
[2] BitBucket repository hosting service, https://bitbucket.org/.
[3] SourceForge source code repository, http://sourceforge.net/.
[4] S. Thummalapenta and T. Xie, "PARSEWeb: a programmer assistant for reusing open source code on the web," in *ASE*, 2007.
[5] V. Raychev, M. T. Vechev, and E. Yahav, "Code completion with statistical language models," in *PLDI*, 2014, p. 44.
[6] T. Gvero and V. Kuncak, "On synthesizing code from free-form queries," EPFL, IC, Tech. Rep. EPFL-REPORT-201606, 2014.
[7] V. Le, S. Gulwani, and Z. Su, "Smartsynth: Synthesizing smartphone automation scripts from natural language," in *MobiSys*, 2013.
[8] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac, "Complete completion using types and weights," in *PLDI*, 2013, pp. 27–38.
[9] D. Jurafsky and J. H. Martin, *Speech and Language Processing*, 2nd ed. Prentice Hall, 2008.
[10] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *ACL*, 2014, pp. 55–60.
[11] M.-C. de Marneffe, B. MacCartney, and C. D. Manning, "Generating typed dependency parses from phrase structure parses," in *LREC*, 2006, pp. 449–454.
[12] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer, "Feature-rich part-of-speech tagging with a cyclic dependency network," in *HLT-NAACL*, 2003.
[13] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, pp. 83–97, 1955.
[14] R. Burkard, M. Dell'Amico, and S. Martello, *Assignment Problems*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2009.
[15] C. Fellbaum, *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.
[16] J. Galenson, P. Reames, R. Bodík, B. Hartmann, and K. Sen, "Codehint: Dynamic and interactive synthesis of code snippets," in *ICSE*, 2014, pp. 653–663.
[17] N. Sahavechaphan and K. Claypool, "Xsnippet: mining for sample code," in *OOPSLA*, 2006.
[18] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *ICSE*, 2005, pp. 117–125.
[19] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *ESEC/SIGSOFT FSE*, 2009, pp. 213–222.
[20] http://www.eclipse.org/recommenders/.
[21] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the api jungle," in *PLDI*, 2005.
[22] A. Cozzie and S. T. King, "Macho: Writing programs with natural language and examples," University of Illinois at Urbana-Champaign, Tech. Rep., 2012.
[23] G. Little and R. C. Miller, "Keyword programming in Java," in *ASE*, 2007, pp. 84–93.
[24] D. Price, E. Riloff, J. L. Zachary, and B. Harvey, "NaturalJava: A natural language interface for programming in Java," in *IUI*, 2000, pp. 207–211.
[25] S. Chatterjee, S. Juvekar, and K. Sen, "SNIFF: A search engine for java using free-form queries," in *FASE*, 2009, pp. 385–400.
[26] S. P. Reiss, "Semantics-based code search," ser. ICSE '09, Washington, DC, USA, 2009, pp. 243–253.
[27] K. T. Stolee, S. Elbaum, and D. Dobos, "Solving the search for source code," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, pp. 26:1–26:45, Jun. 2014.
[28] M. Harman, Y. Jia, and W. B. Langdon, "Babel pidgin: SBSE can grow and graft entirely new functionality into a real world system," in *SSBSE Challenge Track*, 2014, pp. 247–252.